OCAML example: sets as lists

In this section we will do a barebones implementation of sets as lists of unique items. Along the way we will also learn how to work with OCAML's interfaces.

OCAML comes with modules for Set, Map, HashTable and other goodies. You probably want to use those in the long run.

The two files we will be working on are:

- set2.ml[1], the implementation file
- set2.mli[2], the interface file

OCAML forces this separation on us. The .mli file lists the types and functions available. It is similar to the .h files in C.

We are calling our "module" Set2 to avoid conflicts with the included Set module.

Sets as lists

Let us start by taking a look at parts of the interface file:

```
type 'a t

val empty  : 'a t
val member : 'a -> 'a t -> bool
val add    : 'a -> 'a t -> 'a t
val from_list : 'a list -> 'a t
val to_list   : 'a t -> 'a list
```

So we specify here that there is one new type, but we give no more information about it (though we could have). We could have told it for example that 'a t is meant to be a 'a list, but we didn't want to commit ourselves to it: Noone using our "sets" should care that we implemented them via lists. Note that this is a "parametric" type, because of the 'a part there. This is because we can have lots of different "sets": We can have integer sets, of type int Set2.t, floating point number sets, of type float Set2.t, even sets whose elements are themselves sets of integers (this last would be of type int Set2.t Set2.t).

Following that type is a list of "type declarations". So it says for instance that there should be a value called empty of type 'a t. There should also be a value member of type 'a -> 'a t -> bool. This is an arrow type, so we are talking about a function, that takes two arguments, a value and a set, and it tells us if the element is in the set or not.

See if you can guess what the remaining functions do, what arguments they take and what they return.

Let us look at the implementation of these methods, in the "set2.ml" file:

[1] ../ocaml/set2.ml
[2] ../ocaml/set2.mli

```
type 'a t = 'a list

let empty = []
let rec member a lst =
   match lst with
      []       -> false
    | x :: rest -> a = x || member a rest

let add a lst =
   if member a lst
   then lst
   else a :: lst

let rec unique lst =
   match lst with
      [] | x :: [] -> lst
    | x :: rest    -> add x (unique rest)

let from_list lst = unique lst
let to_list   lst = lst
```

Note that here we are specifying exactly what the type 'a t would be, by saying that it is just another name for the list type. This is typically called a *type alias*.

We then have a series of let statements, to implement the various function values declared in the interface file. First off is the implementation of empty, the "empty set". All it is is the empty list, so that's what the third line in our code does.

Next we implement the member function. It simply goes through the list in a recursive fashion. If it finds the element then it returns true. If it reaches the end of the list, then it returns false.

The add function is similarly straightforward. If the element already exists then it does nothing. Otherwise prepends the element.

Next there is a function called unique. This function is not in the interface, so users of our module will not have access to it. Only our other functions can. It is effectively a private method. It is used in our case in the from_list method, to remove any duplicate values that may have been present.

Finally we have the two functions from_list and to_list for converting back and forth between a "set" and a "list". The from_list function needs to take care to remove duplicates. But the to_list function is essentially the identity function, internally at least. For the outside word the input lst is a set and the output lst is a list. For us inside our Set2 module they are the same thing because of how we implemented the 'a t type, so we can use the identity function.

from_list has another interesting implementation, which you can see in a comment in the file. This implementation avoids using the function unique. But first let's discuss the function unique, as its implementation is fairly interesting and typical of more complicated functions.

So unique starts with an input list lst. It intends to remove duplicates. It proceeds as follows, recursively:

- Pattern match on the list.
- If the list has at most 1 element, just return the list since there can be no duplicates.
- If the list has more elements, then recursively call unique on the tail of the list, and store that in a variable rest' (yes, primes are valid parts of a name!).
- Then look at the head of your list, and see if it is in rest'. Prepend it if it is not. (or in our case, call our member function to do that for us)

This function has an important generic pattern, usually called *folding* in this context. You may be familiar with it as an accumulator. Effectively it consists of some key ingredients:

- A list of values, of a certain type 'a, to work with
- A certain initial value, of some other type 'b (an empty list in our example)
- A process for how to take a new value of type 'a, and the so-far-accumulated value 'b, and combine them to get a new value of type 'b.

Given these 3 ingredients, the rest is automatic: We would start with the initial value, then go through each value in our list and combine it into that initial value, to accumulate the final result.

It is a standard accumulator pattern, something you would do in most languages via a for loop and some variable named acc or something that gets updated at every step. In most functional languages we typically have some higher-order functions to do the same thing. Here is how the function signatures look in this case, and they are called fold_left and fold_right. They are part of the List module[3].

```
val fold_left: ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
val fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

So they both take 3 arguments. The first is a function that takes a value of type 'b, which is the accumulator, and a value of type 'a, which is a list element, and returns a value of type 'b. so this is the function taht combines the "current" value with the accumulated values. Then the fold functions further take a value of type 'b, which is the initial value of the accumulator. Lastly they take a list of elements. They then proceed to apply the function to each element in the list combined with the accumulated value, and update the accumulated value. They then return the final accumulated value, which is of type 'b.

What they differ in is the order in which they do their work:

- fold_left would apply the function to the first value in the list, and combine that with the accumulator, then combine that result with the second value in the list and so on.
- fold_right would effectively apply the function to the last element in the list, combined with the accumulator. Then it would feed the result to the previous element on the list, and so on up.

[3] http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

Here is how we could implement these functions:

```
let rec fold_left f acc lst =
    match lst with
       []       -> acc
     | x :: rest -> fold_left f (f acc x) rest

let rec fold_right f lst acc =
    match lst with
        []       -> acc
      | x :: rest -> f x (fold_right f rest)
```

Though they may appear quite similar, they function somewhat differently. Note however, that our function unique can be rewritten as:

```
let unique lst = List.fold_right add lst [];;
```

So to compute unique lst we call the fold_right function, giving it our function add as the update function f, the list lst as the list of elements to process, and an empty list [] as the initial value for the accumulator. Compare the implementation of fold_right with that of unique above and you will see the similarities.

Test: Compare and constrast the following two lines:

```
List.fold_left (fun acc x -> x :: acc) [] [1;2;3];;
List.fold_right (fun x acc -> x :: acc) [1;2;3] [];;
```

Adding more set functions

There are a number of standard set operations that we want to perform. Let us add them to the interface file first:

```
val size : 'a t -> int              (* how many elements *)
val union : 'a t -> 'a t -> 'a t
val inter : 'a t -> 'a t -> 'a t    (* intersection *)
val diff  : 'a t -> 'a t -> 'a t    (* set difference A - B *)
val subset : 'a t -> 'a t -> bool   (* whether A is subset of B *)
val equal : 'a t -> 'a t -> bool
```

We will start with those. Some are simple to implement, since our sets are just lists, for instance:

```
let size set = List.length set
```

For many others we will use our new friends, the fold functions. If you think about it, a lot of the other operations involve going through the elements of the one set, and possibly adding them to the other set or something like that. We can implement most of these via folds:

```
let union set1 set2 = List.fold_left (fun acc x -> add x acc) set1 set2
let inter set1 set2 = List.fold_left (fun acc x -> if member x set2
                                                  then x :: acc
                                                  else acc)
                                     [] set1
let diff set1 set2 = List.fold_left (fun acc x -> if member x set2
```

```
                                                then acc
                                                else x :: acc)
                                        [] set1
let subset set1 set2 = List.fold_left (fun acc x -> acc && member x set2)
                                        true set1
let equal set1 set2 = subset set1 set2 && subset set2 set1
```

It is very easy to get carried away and use folds for almost everything. And in fact you could! But some of these can be written more clearly by using some of the helper methods in the List module. Let's look at some of them:

```
val filter  : ('a -> bool) -> 'a list -> 'a list
val for_all : ('a -> bool) -> 'a list -> bool
val exists  : ('a -> bool) -> 'a list -> bool
```

Let us think about what these do. They all take as first argument a predicate, i.e. a function that given an element of type 'a returns a boolean. Then filter takes such a predicate and a list, and only retains those elements from the list that satisfy the predicate. It is also required to maintain the order.

for_all similarly takes a predicate and a list, and returns whether all elements of the list satisfy the predicate. exists does the same but only asking for at least one element from the list to satisfy the predicate. Special consideration must be given to the empty list: By convention if you like, for_all is true for the empty list, regardless of predicate, and exists is false.

Here is how we could define these functions (they are already defined so we don't really need to). We could have used folds, but it is preferable for for_all and exists to stop as soon as they have enough information (for instance for_all can stop the moment some value fails it). So we will define them with recursive formulas:

```
let rec filter pr lst =
    match lst with
        []        -> []
      | x :: rest -> if p x then x :: filter pr rest
                            else filter pr rest
let rec for_all pr lst =
    match lst with
        []        -> true
      | x :: rest -> p x && for_all pr rest

let rec exists pr lst =
    match lst with
        []        -> false
      | x :: rest -> p x || exists pr rest
```

Anyway, using filter could make inter and diff a whole lot easier to write. Give it a go!